FDTool: a Python application to mine for functional dependencies and candidate keys in tabular data

Matt Buranosky[1] Elmar Stellnberger[2] Emily Pfaff[3]
David Diaz-Sanchez[1] Cavin Ward-Caviness[1]

[1]US Environmental Protection Agency, National Health and Environmental Effects Research Laboratory, Chapel Hill, NC
[2]University of Klagenfurt, Klagenfurt, Austria
[3]University of North Carolina – Chapel Hill, Chapel Hill, NC

Abstract

Functional dependencies and candidate keys are essential for table decomposition, database normalization, and data cleansing. In this paper, we present FDTool, a command line Python application to discover functional dependencies in tabular datasets and infer equivalent attribute sets and candidate keys from them[1].

1 Functional Dependencies

Functional dependencies (FDs) are key to understanding how attributes in a database schema relate to one another. An FD defines a rule constraint between two sets of attributes in a relation[2] $r(U)$, where $U = \{v_1, v_2, \ldots, v_m\}$ is a finite set of attributes [3]. A combination of attributes over a dataset is called a *candidate* [3]. An FD $X \to Y$ asserts that the values of candidate X uniquely determine those of candidate Y [3]. For example, the social security number (SSN) attribute in a dataset of public records functionally determines the first name attribute. Because the FD holds, we write $\{SSN\} \to \{first\_name\}$.

[1]The most recent FDTool source code is available at `https://github.com/USEPA/FDTool`.

[2]Each attribute v_i has a finite domain, written $dom(v_i)$, representing the values that v_i can take on. For a subset $X = \{v_i, \ldots, v_j\}$ of U, we write $dom(X)$ for the Cartesian product of the domains of the individual attributes in X, namely, $dom(X) = dom(v_i) \times \ldots \times dom(v_j)$ [1]. A *relation* r on U, denoted $r(U)$, is a finite set of mappings $\{t_1, \ldots, t_n\}$ from U to $dom(U)$ with the restriction that for each mapping $t \in r(U)$, $t[v_i]$ must be in $dom(v_i)$, $1 \le i \le m$, where $t[v_i]$ denotes the value obtained by restricting the mapping t to v_i. Each mapping t is called a *tuple* and $t(v_i)$ is called the v_i-value of t [2].

Definition 1.1. A *functional dependency* $X \to Y$, where $X, Y \subseteq U$, is satisfied by $r(U)$, if for all pairs of tuples $t_i, t_j \in r(U)$, we have that $t_i[X] = t_j[X]$ implies $t_i[Y] = t_j[Y]$ [1].

In this case, X is the *left-hand side* of an FD, and Y is the *right-hand side* [3]. If Y is not functionally dependent on any proper subset of X, then $X \to Y$ is *minimal* [3]. Minimal FDs are our only concern in rule mining FDs, since all other FDs are logically implied. For instance, if we know $\{SSN\} \to \{first\_name\}$, then we can infer that $\{SSN,\ last\_name\} \to \{first\_name\}$.

1.1 Power set lattice

The search space for FDs can be represented as a *power set lattice* of nonempty attribute combinations.

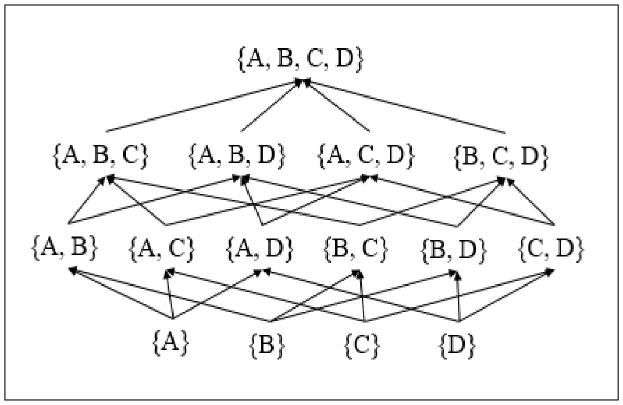


Figure 1: Nonempty combinations of attributes A, B, C, and D by k-level.

Figure 1 gives the nonempty combinations of attributes of a relation $r(U)$ such that $U = \{A, B, C, D\}$. There are $2^n - 1 = 2^4 - 1 = 15$ attribute subsets in the power set lattice [1]. Each combination X of the attributes in U can be the left-hand side of an FD $X \to Y$ such that $X \to Y$ is satisfied by relation $r(U)$ [1]. Since the attribute set itself U trivially determines each one of its subsets, it can be ignored as a candidate. There remain $2^n - 2 = 2^4 - 2 = 14$ nonempty, proper subsets of U that are to be considered candidates.

There are $n \cdot 2^{n-1} - n = 4 \cdot 2^{4-1} - 4 = 28$ edges (or arrows) in the semilattice of the complete search space for FDs in relation $r(U)$ [1]. The size of the search space for FDs is exponentially related to the number of attributes in U. Hence, the search space for FDs increases quite significantly when there is a greater number of attributes in U. For instance, when there are 12 attributes in a relation, the search space for FDs climbs to $24,564$. This gives reason to be cautious of runtime and memory costs when deploying a rule mining algorithm to discover FDs.

1.2 Partition

The algorithms used to discover FDs differ in their approach to navigating the complete search space of a relation. Their candidate pruning methods vary and sometimes the methods used to validate FDs do as well. These differences affect runtime and memory behavior when used to process tables of different dimensions.

A common data structure used to validate FDs is the partition. A partition places tuples that have the same values on an attribute into the same group [3].

Definition 1.2. Let $X \subseteq U$ and let $t_1, \ldots, t_n$ be all the tuples in a relation $r(U)$. The *partition* over X, denoted $\prod_X$, is a set of the groups such that t_i and t_j, $1 \leq i,\, j \leq n$, $i \neq j$, are in the same group if and only if $t_i\,[X] = t_j\,[X]$ [3].

It follows from Definition 1.2 that the *cardinality of the partition* $\mathbf{card}(\prod_A(r))$ is the number of groups in partition $\prod_A$ [1]. The cardinality of the partition offers a quick approach to validating FDs in a dataset.

Theorem 1.1. An FD $X \to Y$ is satisfied by a relation $r(U)$ if and only if $\mathbf{card}(\prod_X) = \mathbf{card}(\prod_{XY})$ [4].

Theorem 1.1 provides an efficient method to check whether an FD $X \to Y$ holds in a relation[3]. Huhtala et al. proved it to support a fast validation method for relations consisting of a large number of tuples [4].

1.3 Closure

Efforts in relational database theory have lead to more runtime and memory efficient methods to check the complete search space of a relation for FDs. In place of needing each arrow in a semi-lattice checked, we can infer the FDs that logically follow from those already discovered. Such FDs are to be discovered as a consequence of *Armstrong's Axioms* [2] and the inference axioms derivable from them [5], which are

– *Reflexivity*: $Y \subseteq X$ implies $X \to Y$;
– *Augmentation*: $X \to Y$ implies $XZ \to YZ$;
– *Transitivity*: $X \to Y$ and $Y \to Z$ imply $X \to Z$;
– *Union*: $X \to Y$ and $X \to Z$ imply $X \to YZ$;

[3]FDTool uses Theorem 1.1 as means to check FDs in the GetFDs module with the *pandas* data analysis library functions *nunique()* and *dropduplicates.count()*.

– *Decomposition*: $X \rightarrow YZ$ implies that $X \rightarrow Y$ and $X \rightarrow Z$.

These axioms signal the distinction between FDs that can be inferred from already discovered FDs, and those that cannot [2]. Exploiting what can be derived from Armstrong's Axioms allows us to avoid having to check many of the candidates in a search space.

Definition 1.3. Let F be a set of functional dependencies over a dataset D and X be a candidate over D. The *closure of candidate* X with respect to F, denoted X^+, is defined as $\{Y \mid X \rightarrow Y$ can be deduced from F by Armstrong's Axioms$\}$ [1].

The *nontrivial closure*[4] *of candidate* X with respect to F is defined as $X^* = X^+ \setminus X$ and written X^* [1]. Definition 1.3 gives room to elegantly define keys. Informally, a key implies that a relation does not have two distinct tuples with the same values for those attributes. Keys uniquely identify all tuple records.

Definition 1.4. Let R be a relational schema and X be a candidate of R over a dataset D. If $X \cup X^* = R$, then X is a *key* [3].

A *candidate key* of a relation is a minimal key for that relation. This means that there is no proper subset Y of candidate key X for which Definition 1.4 holds.

2 Rule mining algorithms

Existing functional dependency algorithms are split between three categories: Difference- and agree-set algorithms (e.g., Dep-Miner, FastFDs), Dependency induction algorithms (e.g., FDEP), and Lattice traversal algorithms (e.g., TANE, FUN, FD_Mine, DFD) [6].

Difference- and agree-set algorithms model the search space of a relation as the cross product of all tuple records [6]. They search for sets of attributes agreeing on the values of certain tuple pairs. Attribute sets only functionally determine other attribute sets whose tuple pairs agree, i.e., *agree-sets* [6] [7]. Then, agree-sets are used to derive all minimal FDs.

Dependency induction algorithms assume a base set of FDs in which each attribute functionally determines each other attribute [6]. While iterating through row data, observations are made that require certain FDs to be removed from the base set and others added to it. These observations are made by comparing tuple pairs based on the equality of their projections. After each record in a

[4]FDTool saves the closure of candidates at each level before releasing it from memory at levels that follow.

dataset is compared, the FDs left in the base set are considered valid, minimal and complete [6].

Lattice traversal algorithms model the search space of a relation as a power set lattice. Most of such algorithms, (i.e., TANE, FUN, FD_Mine) use a level-wise approach to traversing the search space of a relation from the bottom-up [6]. They start by checking[5] for FDs that are singleton sets on the left-hand side and iteratively transition to candidates of greater cardinality.

2.1 Performance

Papenbrock et al. released an experimental comparison of the aforementioned FD discovery algorithms [6]. The seven algorithms were re-implemented in Java based on their original publications and applied to 17 datasets of various dimensions. They found that none of the algorithms are suited to yield the complete result set of FDs from a dataset consisting of 100 columns and 1 million rows [6]. Hence, it is a matter of discretion to choose the algorithm best fitting the dimensions of a dataset.

The experimental results show that lattice traversal algorithms are the least memory efficient, since each k-level[6] can be a factor greater than the size of the previous level [6]. Difference- and agree-set algorithms and dependency induction algorithms perform favorably in memory experiments as a result of their operating directly on data and efficiently storing result sets. Lattice traversal algorithms scale poorly on tables with many columns (≥ 14 columns) due to memory limits [6].

Lattice traversal algorithms are the most effective on datasets with many rows, because their validation method[7] operates on attribute sets as opposed to data [6]. This puts such algorithms in a special position to rule mine clinical and demographic record datasets, which often consist of long sets of participant records. Difference- and agree-set algorithms and dependency induction algorithms commonly reach time limits when applied to datasets of these dimensions ($> 100,000$ rows) [6].

2.2 Lattice traversal algorithms

Lattice traversal algorithms iterate through k-levels represented in a power set lattice. If the lattice is traversed from the bottom-up, we say the algorithm is *level-wise*.

[5]We say that an FD is *checked* when Theorem 1.1 is used to see if it holds or not [3].
[6]Definition 2.1.
[7]Theorem 1.1.

Definition 2.1. Let X_1, X_2, …, X_k, X_{k+1} be $(k{+}1)$ attributes over a database D. If $X_1X_2 \dots X_k \rightarrow X_{k+1}$ is an FD with k attributes on its left hand side, then it is called a *k-level* FD [3].

The search space for FDs is reduced at the end of each iteration using pruning rules. *Pruning rules* check the validity of candidates not yet checked with FDs already discovered and those inferred from Armstrong's Axioms [1]. After a search space is pruned, an *Apriori_Gen* principle generates k-level candidates with the $(k-1)$-level candidates that were not pruned [1].

Apriori_Gen:

– *oneUp*: generates all possible candidates in C_k from those in C_{k-1}.
– *oneDown*: generates all possible candidates in C_{k-1} from those in C_k.

Level-wise lattice traversal algorithms stop iterating after all candidates in a search space are pruned. In this case, *Apriori_Gen* generates the null set $\emptyset$ raising a flag for the algorithm to terminate. This has the effect of shortening runtime to the degree that FDs are discovered and others are inferred.

2.3 Tane

The level-wise lattice traversal algorithms TANE, FUN, and FD_Mine differ in terms of pruning rules. FUN and FD_Mine expand on the pruning rules of TANE. Released by Huhtala et al., TANE prunes a search space on the basis that only minimal and non-trivial[8] FDs need be checked [4]. TANE restricts the right-hand side candidates C^+ for each attribute combination X to the set

$$C^+(X) = \{A \in R \mid \forall B \in X : X \setminus \{A, B\} \rightarrow B \text{ does not hold}\},$$

which contains all the attributes that the set X may still functionally determine [6]. The set C^+ is used in the following pruning rules [6].

- **Minimality pruning:** If an FD $X \setminus A \rightarrow A$ holds, A and all $B \in C^+(X) \setminus X$ can be removed from $C^+(X)$.
- **Right-hand side pruning:** If $C^+(X) = \emptyset$, the attribute combination X can be pruned from the lattice, as there are no more right-hand side candidates for a minimal FD.
- **Key pruning:** If the attribute combination X is a key, it can be pruned from the lattice.

Key pruning implies that all supersets of a key, i.e., *super keys*, can be removed, since they are by definition non-minimal [4].

[8]An FD $X \rightarrow A$ is *non-trivial* if and only if $A \notin X$ [4].

3 FD_Mine

Like TANE and FUN, FD_Mine is structured around the level-wise lattice traversal approach and the aforemented pruning rules. Unlike the other two algorithms, FD_Mine, authored by Yao et al., uses the concept of equivalence as means to more exhaustively prune the search space of a candidate [6]. Informally, attribute sets are equivalent if and only if they are functionally dependent on each other [6].

The proofs demonstrating that no useful information is lost in pruning candidates from equivalent attribute sets are reproduced in this section and were originally developed by Yao et al. [1]. The equivalence pruning method can be derived directly from Armstrong's Axioms.

Definition 3.1. Let X and Y be candidates over a dataset D. If $X \to Y$ and $Y \to X$ hold, then we say that X and Y are an *equivalence* and denote it as $X \leftrightarrow Y$.

After a k-level is fully validated, i.e., each k-level candidate is checked, FD_Mine determines equivalent attribute sets with the FDs already discovered.

Theorem 3.1. Let $X, Y \subseteq U$. If $Y \subseteq X^+$ and $X \subseteq Y^+$, then $X \leftrightarrow Y$ [1].

Proof. Since $X \to X^+$ and $Y \subseteq X^+$, Decomposition implies that $X \to Y$. By a similar argument, $Y \to X$ holds. Because $X \to Y$ and $Y \to X$, we have by definition that $X \leftrightarrow Y$ holds. $\square$

Lemmata 3.2 and 3.3 are derived from Armstrong's Axioms with the assumption of the equivalence $X \leftrightarrow Y$.

Lemma 3.2. Let $W, X, Y, Y', Z \subseteq U$ and $Y \subseteq Y'$. If $X \leftrightarrow Y$ and $XW \to Z$, then $Y'W \to Z$ [1].

Proof. Suppose that $X \leftrightarrow Y$ and $XW \to Z$. This implies that $X \to Y$. By Augmentation, $YW \to XW$. By Transitivity, $YW \to XW$ and $XW \to Z$ give that $YW \to Z$. By Augmentation, $Y' \setminus Y$ can be added to both sides of $YW \to Z$ to give that $YW(Y' \setminus Y) \to Z(Y' \setminus Y)$. By $Y \subset Y'$, we know that $Y'W \to Z(Y' \setminus Y)$. Then, by Decomposition, $Y'W \to Z$. $\square$

Lemma 3.3. Let $W, X, Y, Z \subseteq U$. If $X \leftrightarrow Y$ and $WZ \to X$, then $WZ \to Y$ [1].

Proof. By $X \leftrightarrow Y$, we know that $X \to Y$. By Transitivity, $WZ \to X$ and $X \to Y$ imply $WZ \to Y$. $\square$

Theorem 3.1 checks attribute sets X and Y for the equivalence $X \leftrightarrow Y$. FD_Mine assumes that the attribute set Y is generated before X. By Lemmata 3.2 and 3.3, we know that for equivalence $X \leftrightarrow Y$, no further attribute sets Z such that $Y \subseteq Z$ need be checked [1]. Hence, Y is deleted as a result of the following pruning rule.

- **Equivalence pruning:** If $X \leftrightarrow Y$ is satisfied by relation $r(U)$, then candidate Y can be deleted. [1].

Exploiting the equivalence pruning method leaves FD_Mine in a more aggressive position to prune candidates than TANE. This offers an advantage in terms of runtime and memory behavior [3].

3.1 Non-minimal FDs

The pseudo-code proposed in the second version of FD_Mine [1] will under certain circumstances output non-minimal FDs [6]. FD_Mine references an *Apriori_Gen* method [8] stating that for each pair of candidates $p, q \in C_{k-1}$ the set $p \cup q$ is to be placed in C_k if $\textbf{card}(p \cup q) = k$. Example 1 shows that the *Apriori_Gen* method referenced and utilized by FD_Mine can violate minimality pruning by checking supersets that need not be checked.

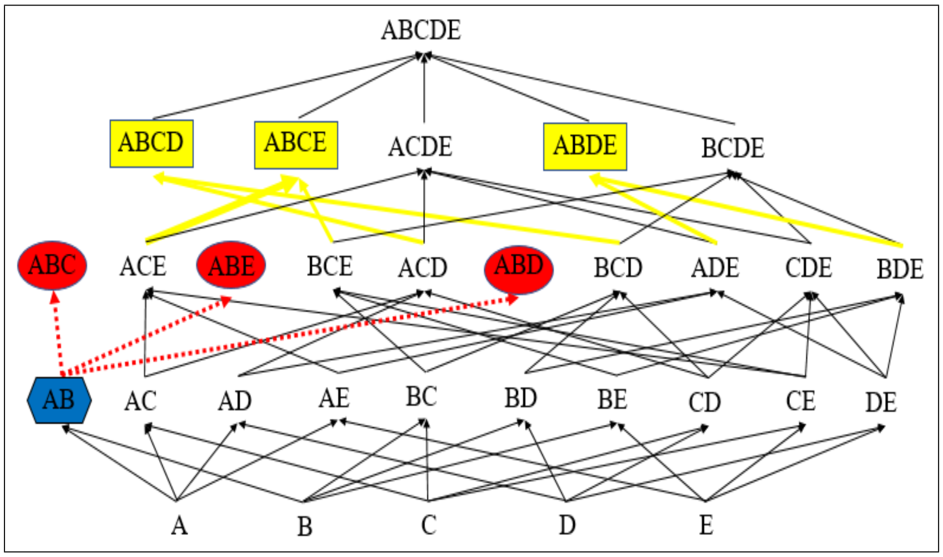


Figure 2: A pruned power set lattice. FD_Mine deletes the candidates ABC, ABE, and ABD (red ovals) as a result of finding the candidate key AB (blue hexagon). It generates supersets of AB (yellow rectangles) at the next level.

Example 1. Let $r(U)$ be a relation such that $U = \{A, B, C, D, E\}$. Suppose that AB is a key and that there are no other FDs in $r(U)$. Since AB is a key, we know by definition that $AB \cup AB^* = U$. Provided this and that there are no other FDs in $r(U)$, the candidates ABC, ABD and ABE are deleted from C_3, and so $C_3 = \mathit{Prune}(\mathit{Apriori\_Gen}(C_2)) = \{ACE,\ BCE,\ ACD,\ BCD,\ ADE,\ CDE,\ BDE\}$ [9]. Then, $C_4 = \{ABCD,\ ABCE,\ ACDE,\ ABDE,\ BCDE\}$. Because it must be that $AB^* = \{C,\ D,\ E\}$, the algorithm validates the FDs $ABCD \rightarrow E$, $ABCE \rightarrow D$, and $ABDE \rightarrow C$. Since E, for example, is functionally dependent on the proper subset $AB \subseteq ABCD$, $ABCD \rightarrow E$ is non-minimal.

The *Apriori_Gen* principle presented in TANE [4] more effectively generates candidate level C_{k+1} from C_k. It requires that C_{k+1} only contains the attribute sets of size $k+1$ which have all their subsets of size k in C_k [4]; i.e.,

$$C_{k+1} = \{X \mid \mathbf{card}(X) = k+1 \text{ and for all } Y \text{ with } Y \subseteq X \text{ and } \mathbf{card}(Y) = k \text{ we have } Y \in C_k\}.$$

In reference to Example 1, this method does not insert the candidate $ABCD$ in C_4, without loss of generality, because $ABC \subseteq ABCD$ but $ABC \notin C_3$. Thus, the non-minimal FD $ABCD \rightarrow E$ is not checked.

Prune(C_k, E, *Closure*) [10][11][12]

```
01 for each $S \in C_k$:
02     for each $X \in oneDown\,[C_k]$:
03         if ($X \subset S$) then:
04             if ($X \in \{Z \mid Y \leftrightarrow Z \in E\}$) then:   # Pruning rule 1
05                 delete $S$ from $C_k$
06             if $S \subset X^+$ then:                                  # Pruning rule 2
07                 delete $S$ from $C_k$
08             $S^+ = S^+ \cup X^*$                                     # Pruning rule 3
09             if $U == S^+$ then:                                       # Pruning rule 4
10                 delete $S$ from $C_k$
11 return $C_k$, $Closure$;
```

FD_Mine will under the circumstance described in Example 1 set closure values incorrectly. In line 2, FD_Mine iterates through C_{k-1}, as opposed to

[9] Since $E = \emptyset$ in this example, we can ignore the argument E in the function $Prune()$. For simplicity's sake, we ignore the argument *Closure*.

[10] Closure = $\{X^+ \mid X \in C_k \vee X \in OneDown\,[C_k]\}$.

[11] Equivalent candidates are stored in E.

[12] All candidates at level k are stored in C_k.

$oneDown\,[C_k]$, which can cause the $Prune()$ function to ignore setting the closure values of certain candidates. In Example 1, FD_Mine does not accurately set the closure $ABCD^*$ to E, since E is not saved to the closure values of the candidates $ACD, BCD \subseteq ABCD$ at the previous level. Iterating through $oneDown\,[C_k]$ sets the closure of a candidate to the union of the closure values of its proper subsets, so that the closure values of deleted candidates are not lost among their supersets.

Properly assigned closure values can allow the algorithm to avoid checking many non-minimal FDs. This is because the $ObtainFDs$ module, i.e., the validation method, only checks[13] the right-hand side attributes v_i for which $v_i \in U \setminus X^+$ [1]. Hence, provided that Pruning rule 3 asserts the equality $ABCD^* = E$, $ABCD \to E$ need not be checked.

4 Experimentation

FDTool was initially created to help decompose datasets of medical records as part of Clinical Archived Records research for Environmental Studies (CARES). CARES currently contains 12 datasets obtained from the medical software firms Epic and Legacy. The attribute count in this database ranges from 4 to 18; the row count ranges from $42,369$ to $8,201,636$.

4.1 Experimental results

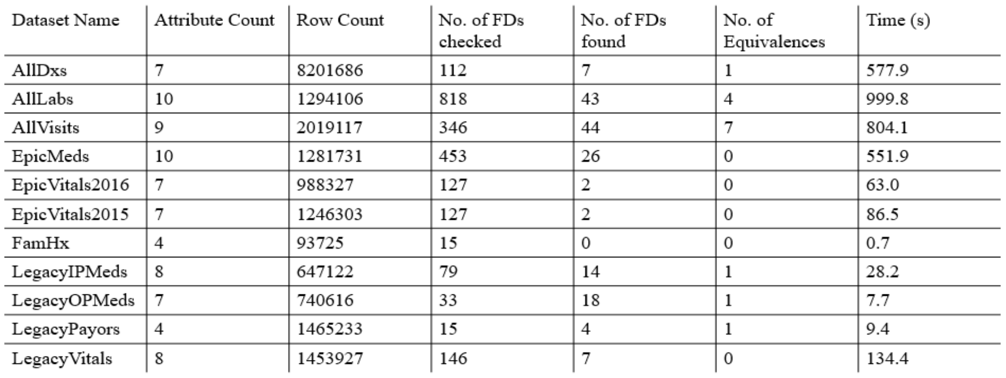
To limit the strain on computational resources, FDTool has a built in time limit of 4 hours. FDTool reaches this preset limit (triggering program termination) when applied to the PatientDemographics dataset (42369 rows $\times$ 18 columns) and the EpicVitals_TobaccoAlcOnly dataset (896962 rows $\times$ 18 columns). The remaining 10 CARES datasets are given in Figure 3[14].

4.2 Experimental summary

The results from Figure 3 show that runtime is primarily determined by the number of attributes in a dataset. For instance, the LegacyPayors dataset (1465233 rows $\times$ 4 columns) has slightly more rows (13% increase) but far fewer attributes (60% decrease) as compared to the AllLabs dataset (1294106 rows $\times$ 10 columns). The runtime of LegacyPayers (9.4 s.) is much less than that of AllLabs (999.8 s.), because AllLabs has many more arrows in its powerset

[13] Assume the left-hand side attribute set X.

[14] **OS**: Windows 10; **Installed memory (RAM)**: 256 GB; **Processor**: Intel Core, 1 CPU; **Clock speed**: 2.19 GHz; **Python:** 2.7.12; **Pandas:** 0.18.1.



Dataset Name	Attribute Count	Row Count	No. of FDs checked	No. of FDs found	No. of Equivalences	Time (s)
AllDxs	7	8201686	112	7	1	577.9
AllLabs	10	1294106	818	43	4	999.8
AllVisits	9	2019117	346	44	7	804.1
EpicMeds	10	1281731	453	26	0	551.9
EpicVitals2016	7	988327	127	2	0	63.0
EpicVitals2015	7	1246303	127	2	0	86.5
FamHx	4	93725	15	0	0	0.7
LegacyIPMeds	8	647122	79	14	1	28.2
LegacyOPMeds	7	740616	33	18	1	7.7
LegacyPayors	4	1465233	15	4	1	9.4
LegacyVitals	8	1453927	146	7	0	134.4

Figure 3: Experimental results of FDTool on 10 CARES datasets, which terminate in less than 4 hours (preset limit).

lattice,

$$n \cdot 2^{n-1} - n = 10 \cdot 2^{10-1} - 10 = 5110,$$

than does LegacyPayers (28). Hence, FDTool has more FDs to check when applied to AllLabs. It is clear that the attribute count of a dataset has a much greater effect on the runtime of FDTool than does row count.

Many of the arrows in the powerset lattice of a candidate are pruned by FDTool. AllLabs has 5110 arrows in its powerset lattice. However, FDTool only checks 818 FDs, as there are many inferred from the 43 FDs found. This follows from the *Prune()* function, which deletes many of the candidates to check partially as a result of mining 4 equivalent attribute sets. FDTool terminates after 5 k-levels when applied to AllLabs.

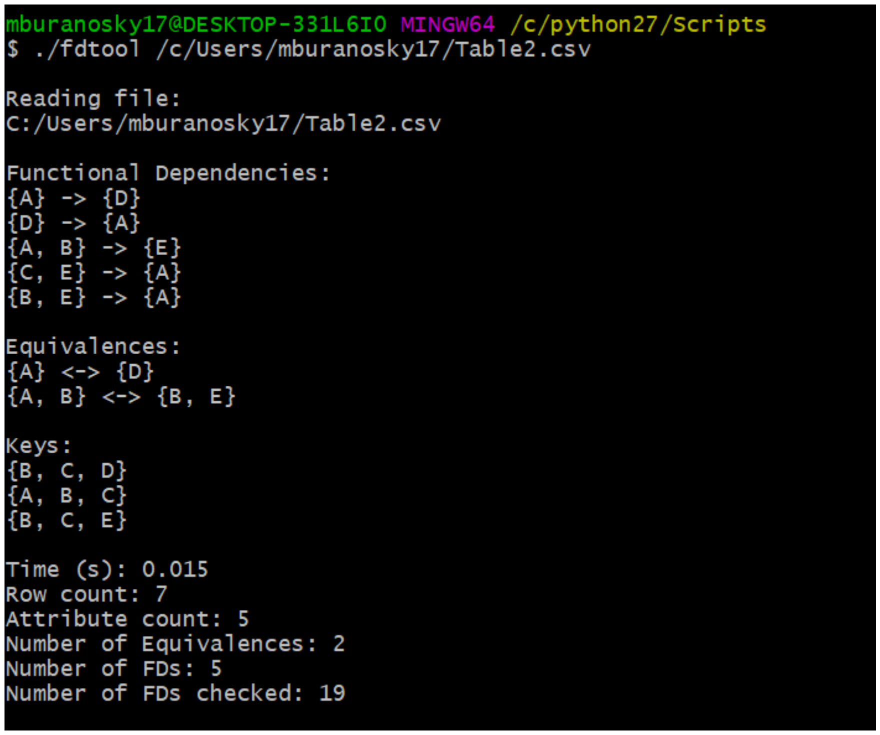
5 Operation

FDTool is a command line Python application executed with the following statement: `$ fdtool /path/to/file` [15]. For Windows users, this is to be run from the directory in which the executable `fdtool.exe` resides, which will likely be `C:\Python27\Scripts` for those installing with `pip install fdtool`. For other systems, installation automatically inserts the file path to the fdtool command in the PATH variable. `/path/to/file` is the absolute or relative path to a .txt, .csv, or .pkl file containing a tabular dataset. If the data file has the extension .txt or .csv, FDTool detects the following separators: comma (`','`),

[15] Edit `FDTool/fdtool/config.py` prior to building setup with `python setup.py install` to change preset time limit or max k-level.

bar (‘|’), semicolon (‘;’), colon (‘:’), and tilde (‘∼’). The data is read in as a Pandas data frame[16].

FDTool provides the user with the minimal FDs, equivalent attribute sets and candidate keys mined from a dataset. This is given with the time (s) it takes for the code to terminate (after reading in data), the row count and attribute count of the data, the number of FDs and equivalent attribute sets found, and the number of FDs checked. This is printed on the terminal after the code is executed as shown in Figure 4. The information is saved to a .FD_Info.txt file.



```
mburanosky17@DESKTOP-331L6I0 MINGW64 /c/python27/Scripts
$ ./fdtool /c/Users/mburanosky17/Table2.csv

Reading file:
C:/Users/mburanosky17/Table2.csv

Functional Dependencies:
{A} -> {D}
{D} -> {A}
{A, B} -> {E}
{C, E} -> {A}
{B, E} -> {A}

Equivalences:
{A} <-> {D}
{A, B} <-> {B, E}

Keys:
{B, C, D}
{A, B, C}
{B, C, E}

Time (s): 0.015
Row count: 7
Attribute count: 5
Number of Equivalences: 2
Number of FDs: 5
Number of FDs checked: 19
```

Figure 4: Printed output of FDTool.exe.

5.1 Implementation

FDTool is a Python based re-implementation of the FD_Mine algorithm with additional features added to automate typical processes in database architecture. FD_Mine was published in two papers with more detail given to the scientific concepts used in algorithms of its kind [3] [1]. Yao et al. released two versions

[16]The data is read in with the Pandas function *read_csv()*, which is subject to the usual spacing errors associated with reading in delimiter-separated values.

of FD_Mine with different structures but making use of the same theoretical foundation [6]. Their work is fully supported in mathematical proofs of the pruning rules used [1]. FDTool was coded[17] with special attention given to the pseudo-code presented in the second version of FD_Mine [1].

The Python script `dbschema.py` in `FDTool/fdtool/modules/dbschema` is taken from *dbschemacmd* (`https://www.elstel.org/database/dbschemacmd.html.en`): a tool for database schema normalization working on functional dependencies [9]. It is used to take sets of FDs and infer candidate keys from them. The operation first assigns the left-hand side attribute combinations of a set of FDs to dictionary keys and their closures to the corresponding values. It then reduces the set of FDs to a minimum coverage[18]. Candidate keys are assembled using the minimum coverage and closure structure by adding attributes to key candidates until each minimal attribute set X for which $X^+ = U$ is found. Details on the dbschema operations are described in `FDTool/fdtool/modules/dbschema/Docs`.

5.2 Software availability

1. FDTool is available from the Python Package Index (PyPi; `https://pypi.python.org`) via: pip install fdtool.
2. Latest source code: `https://github.com/USEPA/FDTool.git`.
3. License: CC0 1.0 Universal (`https://creativecommons.org/publicdomain/zero/1.0/legalcode`). Module `FDTool/fdtool/modules/dbschema` released under a modified C-FSL license.
4. Dependencies:
    - 4.1. Python2 (`https://www.python.org/`), recommended version 2.7.8 or later.
    - 4.2. Pandas data analysis library (`https://pandas.pydata.org/`) via: pip install pandas.

5.3 Future development

We want to improve its performance so that FDTool is better equipped to handle datasets of different dimensions. Using the dependency induction algorithm FDEP, the reach of FDTool could be extended to datasets with fewer rows and more than 100 columns [6]. This might also require upgrading the source code

[17]FDTool was tested regularly throughout the implementation process so as to accomodate to changes made to improve runtime and memory behavior.

[18]A set of FDs F is a *coverage* of another set of FDs G if every FD in G can be inferred from F; i.e., $G^+ \subseteq F^+$ [10]. F is a *minimum coverage* of G if F is the smallest set of FDs that covers G [10].

with multicore processing methods, such as a Java API, to reduce runtime and avoid reaching memory limits. A formal proof of the the dbschema operations is also desired.

Another goal is to increase the functionality provided by FDTool. This would mean implementing the pen and paper methods typically used to normalize relational schema and decompose tables. Our intent is to incorporate these changes in newer versions of FDTool, released at regular periods, so as to develop it as Python software that could automate much of what is done in the database design process.

References

[1] H. Yao and H. J. Hamilton, "Mining functional dependencies from data," *Data Min. Knowl. Discov.*, vol. 16, pp. 197–219, Apr. 2008.

[2] D. Maier, *Theory of Relational Databases.* Computer Science Pr, 1983.

[3] H. Yao, H. Hamilton, and C. Butz, "Fd_mine: Discovering functional dependencies in a database using equivalences.," 01 2002.

[4] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen, "Tane: An efficient algorithm for discovering functional and approximate dependencies," 1999.

[5] R. Ramakrishnan and J. Gehrke, *Database Management Systems.* New York, NY, USA: McGraw-Hill, Inc., 2nd ed., 2000.

[6] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann, "Functional dependency discovery: An experimental evaluation of seven algorithms," *Proc. VLDB Endow.*, vol. 8, pp. 1082–1093, June 2015.

[7] N. Asghar and A. Ghenai, "Automatic discovery of functional dependencies and conditional functional dependencies : A comparative study," 2015.

[8] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A. I. Verkamo, *et al.*, "Fast discovery of association rules.," *Advances in knowledge discovery and data mining*, vol. 12, no. 1, pp. 307–328, 1996.

[9] R. Elmasri and S. Navathe, *Database Systems: Models, Languages, Design, and Application Programming.* Pearson, 2011.

[10] R. Soule, “Functional dependencies and finding a minimal cover,” 2014.